Tecniche di Programmazione Funzionale e Imperativa

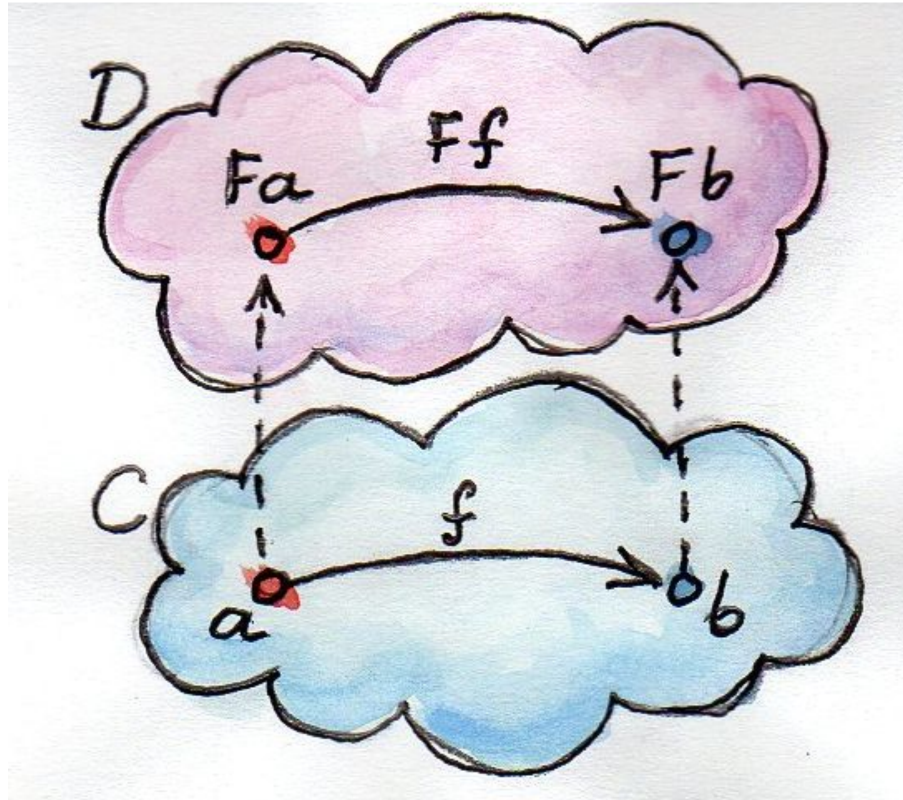
Ivano Salvo

Funtori e Applicativi

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **15**, 12 aprile 2022



Lezione 15a
Funtori

More Maps

Abbiamo visto come la **`map`** e **`foldr`** generalizzino un gran numero di computazioni sulle liste.

Ora vedremo come il concetto di **`map`** si generalizzi a tutti i costruttori di tipo in modo naturale.

Avendo i costruttori del tipo T che da oggetti di tipo a permette di costruire oggetti di tipo $T\ a$ allora da ogni funzione $f :: a \rightarrow b$ posso ottenere una funzione $T\ f :: T\ a \rightarrow T\ b$.

Questo corrisponde al concetto categoriale di funtore.

$$\begin{array}{ccc} Ta & \xrightarrow{T f} & Tb \\ \uparrow & = & \uparrow \\ a & \xrightarrow[f]{} & b \end{array}$$

Per esempio, nelle liste T è dato da **`:`** e **`[]`**

Classe Functor

La classe `Functor` richiede che sia definita una funzione `fmap` che rende **commutativo il diagramma** visto prima.

Osservare che il tipo di `fmap` non è solo parametrico nei tipi `a` e `b`, ma è **parametrico rispetto a un costruttore di tipo** `t`.

Che `t` debba essere un costruttore di tipo, viene inferito dal fatto che **`t` si applica ad altri tipi**.

```
 -- definizione della classe Functor
class Functor t where
   fmap :: (a -> b) -> t a -> t b
```

Esempi

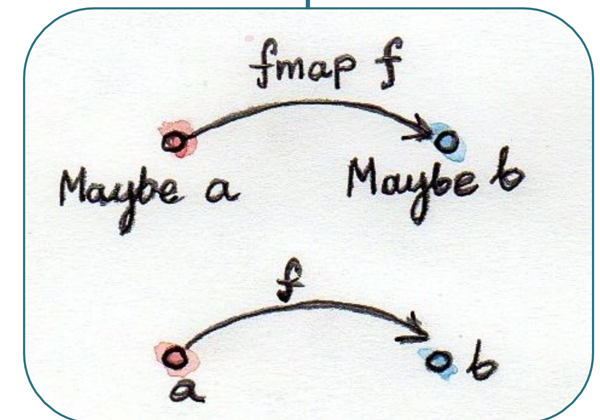
Ovviamente le **liste** sono istanze della classe **`Functor`**

e ovviamente **`fmap`** è proprio la **`map`** già nota.

Possiamo definire **`fmap`** in modo naturale su altri tipi, come ad esempio il costruttore di tipo **`Maybe`**.

O sugli **alberi binari**.

```
 -- dichiaro le liste istanze di Functor
istance Functor [] where
   fmap = map
 -- oppure Maybe
istance Functor Maybe where
   fmap f Nothing  = Nothing
   fmap f (Just x) = Just (f x)
 -- oppure BinTree
istance Functor BinTree where
   fmap f (F x) = F (f x)
   fmap f (R r lft rgt) =
      R (f r) (fmap f lft)(fmap f rgt)
```



Generic Programming

Dichiarare un tipo istanza di una classe permette come sempre di fare del **Generic Programming** tipico, se volete, dei **linguaggi Object Oriented** che sfrutta una certa "**convenzione sui nomi**".

Vediamo un esempio banale, ma molto illuminante su come si possano definire funzioni generiche in (`Functor t`).

```
inc :: Functor t => t Int -> t Int
inc = fmap (+1)
 -- lo posso applicare a qualunque istanza
 -- di Functor
> inc (Just 1)
Just 2
> inc [1,2,3,4,5]
[2,3,4,5,6]
> inc R 2 (F 1) (F 2)
R 3 (F 2)(F 3)
```

inc di fatto è overloaded via overloading di fmap

Functor Laws

Affinchè il diagramma commuti, deve essere vero che:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Queste leggi **dovrebbero essere soddisfatte** da **qualunque** cosa sia definita come **funtore**, ma ovviamente **non possono essere verificate dal type-checker**, ma sono responsabilità del programmatore.

Ad esempio, la seguente definizione tipa, ma non soddisfa le leggi funtoriali (è una sorta di funtore reverse…).

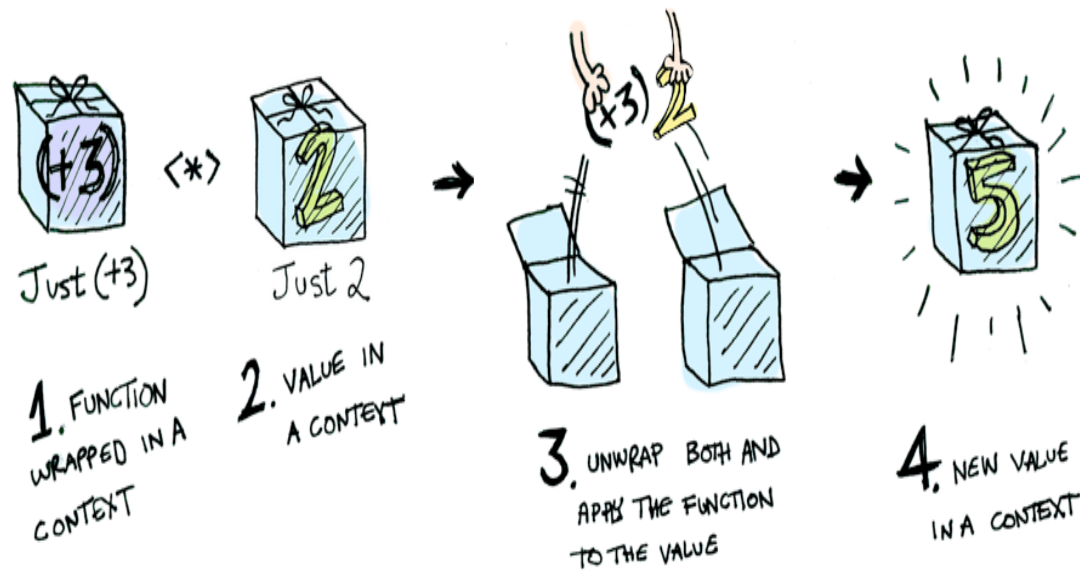
```
istance Functor [] where
   fmap f []     = []
   fmap f (x:xs) = fmap f xs ++ [f x]

 -- infatti non rispetta l'identità
> fmap (\x -> x) [1,2]
[2,1]

 -- e neanche la composizione
> fmap (not . even) [1,2]
[False, True]

> fmap not . fmap even [1,2]
[True, False]
```

fmap che rovescia la lista su cui si applica



Lezione 15b

Funtori Applicativi

More More More ...Maps

Abbiamo visto come generalizzare a tutti i tipi la nozione di **`map`**, attraverso la classe **`Functor`**.

Abbiamo visto che **`zipWith`** **è una forma di** **`map`** **binaria**.

È naturale considerare un caso di generalizzato in cui si possono considerare funzioni **`zipWith`** di **qualsiasi arità**.

Ecco quale potrebbe essere questa gerarchia di funzioni:

```
fmap0 :: a -> t a
fmap1 :: (a -> b) -> t a -> t b
fmap2 :: (a -> b -> c) -> t a -> t b -> t c
fmap3 :: (a -> b -> c -> d) -> t a -> t b -> t c -> t d
```

Ovviamente, non vogliamo scrivere una classe `Functor`$_0$, `Functor`$_1$, `Functor`$_2$, `Functor`$_3$, …

Esempio: Trasposta di una Matrice

[McBride, Paterson, *Applicative Programming with Effects*, JFP **18**, 2007]

Riscriviamo la trasposta di una matrice, in modo un po' più gradevole (se ricordate, usavo 2 casi base, uno per una singola colonna e uno per 0 colonne). Ma si può fare in modo più elegante… ricordando anche la vecchia amica **`applyL`**.

Notare le due applicazioni di **`repeat`**.

```
transpose :: [[a]] -> [[a]]
transpose    []     = repeat []
transpose (xs:xss) =
   repeat (:) `zApp` xs `zApp` (transpose xss)
-- zApp (zApp (repeat (:)) xs) (transpose xss)
-- repeat crea infinite
-- copie dell'argomento
repeat : a -> [a]
repeat x = x : repeat x
-- zApp è applyL: zipWith f = repeat f . zApp
zApp : [a -> b] -> [a] -> [b]
zApp (f:fs) (x:xs)  = f x : zApp fs xs
zApp    _    _      = []
```

map, zipWith & repeat, zApp

Osserviamo che **`zApp`** è una **forma generale di applicazione** che **'commuta'** in qualche senso con il **costruttore del tipo lista** e generalizza sia il comportamento di **`fmap`** che di **`zipWith`**.

Infatti:

```
map f xs = repeat f `zApp` xs (map f = repeat f . zApp)
    zipWith f xs ys = repeat f `zApp` xs `zApp` ys
```

Più in generale:

$$fmap_n :: (a_1 \to a_2 \to \ldots \to a_n \to b) \to [a_1] \to [a_2] \to \ldots \to [a_n] \to b$$

$$fmap_n\ f\ xs_1\ xs_2 \ldots xs_n = repeat\ f\ `zApp`\ xs_1\ `zApp`\ xs_2\ `zApp` \ldots xs_n$$

La definizione di map in termini di **`repeat`** e **`zApp`** dimostra che questo schema di programma generalizza la nozione di funtore, come vedremo presto.

Esempio: Valutazione di Espressioni

Vediamo un esempio analogo: scriviamo un valutatore di semplici espressioni con *somme* e *variabili*. Le variabili necessitano di avere un *ambiente* di valutazione in cui ritrovare il valore associato alla variabile in un certo momento.

Anche qui, l'idea è di evitare di dover trasportare gli ambienti da una parte all'altra.

L'idea è simile a quella di **`zApp`**…

```
data Exp v = Var v | Val Int | Add Exp Exp

eval : Exp v -> Env v -> Int
eval (Var x) env = fetch x env -- trova il valore di x
eval (Val i) env = i
eval (Add e1 e2) env = eval e1 env + eval e2 env
```

Esempio: Valutazione di Espressioni

L'ambiente di valutazione viene 'distribuito' da **S**, e dimenticato da **K** (quando non serve).

[Il parametro *env* esiste ancora, come si vede dal tipo di eval, ma possiamo non occuparcene grazie alla η-regola (usata prima nella definizione di `map` con `repeat` e `zApp`]: $F =_\eta \lambda x.\ Fx$ (se $x \notin FV(F)$)]

```
eval : Exp v -> Env v -> Int
eval (Var x) = fetch x -- trova il valore di x
eval (Val i) = K i
eval (Add e1 e2) = K (+) `S` eval e1 `S` eval e2

# dove K e S sono i combinatori noti:
K x y = x
S x y z = x z (y z)
```

Esempio: Maybe

L'idea è quella di considerare una forma astratta di applicazione che generalizza l'usuale applicazione di funzioni.

Vorremmo ottenere effetti come questo sul tipo **`Maybe`**, che permettono di propagare eccezioni senza dover 'scartare' e 'reincartare' i risultati dal tipo **`Maybe`**.

E poi generalizzare a un numero arbitrario di argomenti senza scrivere le classi **$\texttt{Functor}_0$**, **$\texttt{Functor}_1$**, **$\texttt{Functor}_2$**, **$\texttt{Functor}_3$**…

Vedremo nel seguito come ottenere questo effetto usando solo 2 operatori:

`pure``:: a->t a` (che corrisponde a $\texttt{fmap}_0$) e

`<*>``:: t(a->b) -> t a -> t b`

(che permette di ottenere $\texttt{fmap}_{n+1}$ da $\texttt{fmap}_n$)

```
 -- propagare risultati corretti con eccezioni
> fmap2 (+) (Just 2) (Just 3)
Just 5
```

… e quindi

```
fmap₀ = pure
fmap₁ g x = pure g <*> x
fmap₂ g x y = pure g <*> x <*> y
```

Questo si ottiene con una **estensione della classe** `Functor` come sotto riportato.
Vediamo anche l'esempio di `Maybe`.

```
 -- dichiarazione che di Applicative
 -- come classe derivata da Functor
class Functor t => Applicative t where
   pure   :: a -> t a
   (<*>):: t (a -> b) -> t a -> t b

 -- Definizione di pure e <*> in Maybe
istance Applicative Maybe where
   pure = Just
 -- pure x = (Just x)
   Nothing <*> _    = Nothing
   (Just g)<*> mx = fmap g mx
```

Un semplice esempio

Vediamo anche l'esempio di `Maybe`.
Osservate che <*> associa a **sinistra**.

```
 -- vediamo una semplice riduzione:
pure (+) <*> (Just 2) <*> (Just 3)
    → {pure}
(Just (+)) <*> (Just 2) <*> (Just 3)
    → { <*> }
fmap (+) (Just 2) <*> (Just 3)
    → { fmap }
(Just  (+2)) <*> (Just 3)
    → { <*> }
fmap (+2) (Just 3)
    → { fmap }
(Just  (+5))
```

Vediamo sulle liste

La definizione canonica sulle liste in **`Prelude`** **non porta tuttavia** **`pure f <*> xs <*> ys`** a essere la **`zipWith f xs ys`**!

L'idea è (come in **`Maybe`**) di modellare una sorta di **nondeterminismo** per cui una lista è **una sequenza di risultati possibili**, e applicando una lista di funzioni ad esse, produce **tutte le possibili applicazioni** delle funzioni **a tutti i risultati**.

```
-- Definizione di pure e <*> in []
istance Applicative [] where
    -- pure : a -> [a]
    pure x = [x]

    -- <*> : [a -> b] -> [a] -> [b]
    gs <*> xs = [g x | g <-gs, x<-xs]

> pure (*) <*> [1,2] <*> [3,4]
[3, 4, 6, 8]
 -- e non [3, 8]!!
```

zipList

Esiste una definizione alternativa dei funtori applicativi sulle liste.

Innanzitutto vediamo la keyword **newtype** che serve a ’wrappare’ (incartare) un tipo dentro un altro tipo **con un solo costruttore unario** (qui **l’equivalenza** diventa **per nome**, per cui `ZipList` non è equivalente a (o sinonimo di) `[]`).

Ecco un vecchio amico (**`applyList`**!)

```
-- altro tipo list:
newType ZipList a = Z [a]

instance Applicative ZipList where
   -- pure : a -> [a]
   pure x = Z (repeat x) -- repeat x = x : repeat x

   -- <*> : [a -> b] -> [a] -> [b]
   Z fs <*> Z xs = Z (zipWith (\f x -> f x) fs xs)
   -- Z fs <*> Z xs = Z zapp fs xs
   -- è equivalente ad applyList, ma ovviamente:
   -- pure f <*> Z xs <*> Z ys = Z zipWith f xs ys
```

Applicative Laws

Ecco le leggi (scritte in Haskell) che devono essere soddisfatte dagli applicativi:

1. `<*>` **preserva l'identità**, come si conviene a una forma di applicazione!
2. `<*>` **preserva l'applicazione di funzioni**, nel senso che 'distribuisce sull'applicazione' trasformando l'applicazione usuale (questa è ancora una **forma di funtorialità**)
3. Non è importante l'ordine con cui si fa l'embedding
4. La terza è una forma di **associatività di `<*>`** , analoga a quella della composizione di funzioni (`(.)` si può liftare a livello applicativo)

```
pure id <*> x = x

pure (g x) = pure g <*> pure x

x <*> pure y = pure (\g -> g y) <*> x

x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

Lezione 15

That's all Folks…

Grazie per l'attenzione…

…Domande?